ShapeOp - A Robust and Extensible Geometric Modelling Paradigm

Mario Deuss, Anders Holden Deleuran, Sofien Bouaziz, Bailin Deng, Daniel Piker, Mark Pauly

Abstract We present ShapeOp, a robust and extensible geometric modelling paradigm. ShapeOp builds on top of the state-of-the-art physics solver (Bouaziz et al 2014). We discuss the main theoretical advantages of the underlying solver and how this influences our modelling paradigm. We provide an efficient open-source C++ implementation (www.shapeop.org) together with scripting interfaces to enable ShapeOp in Rhino/Grasshopper and potentially other tools. This implementation can also act as a template for future integration of computer graphics research. To evaluate the potential of ShapeOp we present various examples using our implementation and discuss potential implications on the design process.

Key words: Modelling and Design of Behaviour

Mario Deuss
École Polytechnique Fédérale de Lausanne, Switzerland. e-mail: mario.deuss@epfl.ch

Anders Holden Deleuran
CITA, Royal Danish Academy of Fine Arts, School of Architecture, Copenhagen, Denmark. e-mail: adel@kadk.dk

Sofien Bouaziz
École Polytechnique Fédérale de Lausanne, Switzerland. e-mail: sofien.bouaziz@epfl.ch

Bailin Deng
École Polytechnique Fédérale de Lausanne, Switzerland. e-mail: bailin.deng@epfl.ch

Daniel Piker
Robert McNeel & Associates, London, United Kingdom.e-mail: danielpiker@gmail.com

Mark Pauly
École Polytechnique Fédérale de Lausanne, Switzerland. e-mail: mark.pauly@epfl.ch

1 Introduction

Under the well established geometric modelling paradigms such as constructive solid geometry or spline-based modelling, polygonal mesh modelling yields a good trade-off between expressibility - its many degrees of freedom allow to approximate an arbitrary design - and computational effort - its inherent linear interpolation reduces mathematical complexity. This has led to the development of various form-finding and modelling tools for the exploration of shape spaces of polygonal meshes. In our context we consider a shape space as a set of all designs that respect given geometric constraints dictated by aesthetic, fabrication and cost requirements. See Fig. 6 in (Bouaziz et al 2012) for an example of a shape space.

Shape space exploration is typically facilitated by an optimization algorithm that negotiates a large number of complex and possibly conflicting constraints to satisfy the design goals. Numerical solvers for constraint satisfaction therefore play a fundamental role in shape exploration environments. A number of requirements on these solvers are essential for an effective design process, such as numerical robustness, computational efficiency, flexibility to handle a diverse set of design constraints, and extensibility to adapt to new design environments.

Existing shape exploration methods are often restricted by inherent limitations of their optimization approach. They might be tailored to a specific set of constraints, for example planarity of polygons, which can limit design flexibility. Some of them exhibit numerical instabilities or slow convergence, which makes interactive modelling cumbersome. Last but not least, they are often closed, monolithic software, which makes adaptations or extensions in new design tasks difficult. We propose a new computational approach to geometric modelling and design that alleviates these limitations.

We adopt the physics solver proposed in (Bouaziz et al 2014) that integrates a variety of constraints, dynamics and handle-based shape space exploration, and add projective constraints described in (Bouaziz et al 2012). We refer to the combination as ShapeOp. In this paper we evaluate the potential of ShapeOp for design in a number of examples using Rhino/Grasshopper as a graphical user interface. We also discuss and provide our implementation of ShapeOp, which effectively bridges the gap between computer graphics research and practical computational design, and acts as an open-source template for making research available. ShapeOp can also act as a building block for algorithms exploring further aspects of the shape space, e.g. adaptive meshing, evolutionary optimization and automatic constraint selection. The contribution of this paper is three-fold:

1. We propose ShapeOp, a state-of-the-art unified and extensible constraint solver based on the latest research in computer science, and make it accessible to the architectural modelling community.

2. We describe and provide an efficient C++ open-source implementation of ShapeOp and an integration into Rhino/Grasshopper using Python/ctypes.
3. We highlight design applications and demonstrate the extensibility of ShapeOp in various examples.

2 Related Work

The two papers (Bouaziz et al 2012) and (Bouaziz et al 2014) provide a thorough discussion about previous work related to ShapeOp. In the following, we focus on related work in the domain of constraint optimization and form-finding.

For computational design, numerical optimization is a fundamental tool as it allows multiple requirements to be incorporated in the design process. For example, in architectural geometry, design shapes are often optimized according to certain geometric constraints that correspond to fabrication requirements (Pottmann et al 2015). Typically, a non-linear least squares problem is formulated, with each residual term corresponding to one constraint, and solved using standard solvers such as Gauss-Newton.

Numerical optimizations as discussed above can be time-consuming, due to the need for evaluating the Jacobian and solving a different linear system in each iteration of the solver (Nocedal and Wright 2006). In comparison, each iteration of ShapeOp only involves parallel evaluation of projection operators, as well as the solution of a pre-factorized linear system. Recently, Tang et al (2014) propose a form-finding technique for polyhedral meshes, with much better performance than classical non-linear least squares formulations. However, their approach still relies on solving different linear systems in each iteration, resulting in poor performance for meshes with more than a few thousand vertices. On the contrary, the fixed linear system in ShapeOp makes it suitable even for large models.

For form-finding, one popular approach is to model the shape as a system of nodes subject to internal and external forces, and to compute the final shape as an equilibrium state of the system (Day 1965; Kilian and Ochsendorf 2005; Attar et al 2009; Senatore and Piker 2015). For example, Kilian and Ochsendorf (2005) use particle-spring systems for finding structural forms composing only axial forces. Using an implicit Runge-Kutta solver for computing the equilibrium state, their method allows the user to interact with the simulation while it is running. Such force-based approach is also adopted in Kangaroo, a live physics engine built on top of Grasshopper (Piker 2013). By modelling geometric constraints as forces, Kangaroo can perform not only form-finding and physics simulation, but also constraint solving and optimization, making it a popular tool among architects.

Although such force systems are intuitive to set up, it is challenging to simulate their behavior in an efficient, accurate, and stable way (Witkin and Baraff 1997). Implicit solvers allow for a large time step and require fewer iterations, but each iteration can be quite costly to compute since it requires solving a system of algebraic equations. Also, adding new forces requires the derivation of a Jacobian, making them more difficult to extend. Explicit solvers involve lower computational cost for each iteration, but require smaller step sizes to produce stable results, which can result in a large number of iterations. For example, one issue of Kangaroo as presented in (Piker 2013) is that the simulation can explode for highly stiff problems, since such problems require a step size much smaller than the default value; as a result, it is difficult to compute a shape that satisfies the given constraints exactly, because this will require large forces for the constraints and lead to very stiff systems.

Unlike such force-based approaches, ShapeOp computes the equilibrium state of a system by minimizing a potential energy that incorporates physical forces as well as geometric constraints. Using the carefully designed numerical solver in ShapeOp, a stable solution can be computed in a small number of iterations with low computational cost, achieving better stability and efficiency than force-based solvers. Another benefit of ShapeOp is that it is fully open-source, with bindings for many languages including C, C++, C#, Java, and Python. This makes it easy to use ShapeOp from different programming environments, and to extend and adapt the codes according to specific needs.

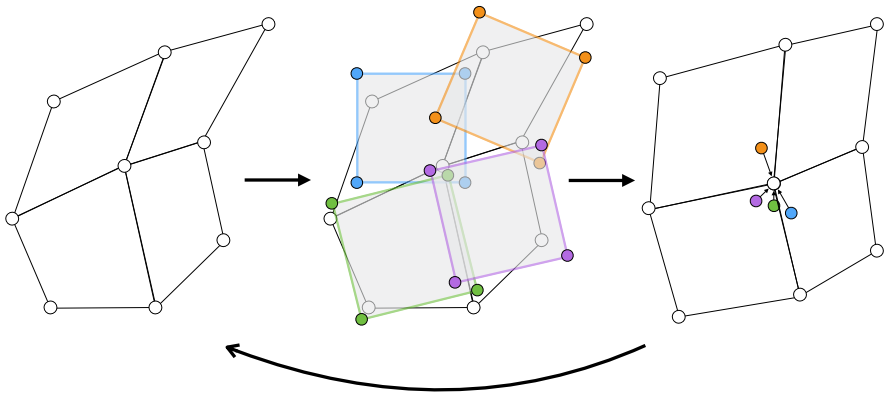

Fig. 1 A quad mesh constrained to consist of squares illustrating the ShapeOp solver. Left: Initial configuration. Middle: Local step - Projecting each quad onto its closest square. Right: Global step - Joining the individual projections by a global minimization. The resulting mesh is then used as initial configuration and the solver iterates.

3 Solver

ShapeOp is a physics engine as well as an optimization tool, designed for a set of points that are subject to physical and geometric constraints. In dynamic mode ShapeOp simulates physics by preserving momentum. In static mode ShapeOp optimizes for an equilibrium solution, which converges considerably faster due to the absence of oscillations induced by momentum.

For constrained optimization, ShapeOp adopts the iterative solver of (Bouaziz et al 2014), which models physical potentials as well as geometric constraints including the ones presented in (Bouaziz et al 2012) in a unified manner. Each iteration of the solver consists of a local step and a global step (see Fig. 1):

Local Step A candidate shape is computed for each set of points that are commonly influenced by a constraint. For a geometric constraint, this amounts to fitting to the points a shape that satisfies the constraint. For a physical constraint, this reduces to finding the closest point positions that have zero physical potential value.

Global Step The candidate shapes computed in the local step are incompatible. The global step solves for a new set of consistent point positions such that each set of points subject to a common constraint are as close as possible to the corresponding candidate shape (see Fig. 1 right).

By repeating the above steps, the overall constraint violation is decreased in each iteration, and the mesh converges to a shape that satisfies all physical and geometric constraints as much as possible. Moreover, each iteration can be run very efficiently: in the local step different constraints can be handled in parallel, while in the global step we only need to solve a linear system with a fixed matrix.

For dynamics simulation, ShapeOp uses the implicit Euler integration scheme from (Bouaziz et al 2014), where at each integration step the physical and geometric constraints are resolved using the above local-global solver. Thanks to the efficiency of the local-global solver, ShapeOp benefits from the stability of implicit integration, with significantly lower computational cost than traditional implicit Euler solvers. ShapeOp also allows defining external forces such as wind and gravity. For more information, please refer to the paper (Bouaziz et al 2014).

4 Projections

Central to the constrained optimization solver in ShapeOp are the so-called projection operators, which are used to compute the candidate shapes in the local step. Specifically, given a set of points that are subject to a constraint, the projection operator finds the closest point positions that satisfy the constraint. A new constraint can be

added easily to ShapeOp, as long as its projection operator is provided. No changes to global step of the solver are necessary to add a constraint.

A simple example of a constraint is the closeness constraint: It is satisfied if the constrained point x coincides with a prescribed position c. Since the only way to satisfy the closeness constraint is by setting x equal to c , the projection $P(\cdot)$ simply is given by $P(x) = c$. The documentation of ShapeOp also provides a step-by-step tutorial on an orientation constraint, from formulation to implementation. See Table 1 for all constraints implemented in ShapeOp.

Table 1 Constraints implemented in ShapeOp.

Constraint	Description
Edge, Triangle, Tetrahedron Strain	Bounds the strain with respect to its initial configuration (Bouaziz et al 2014, § 5.1).
Area, Volume	Bounds the area (volume) of a triangle (tetrahedron) (Bouaziz et al 2014, § 5.2).
Bending	Bounds the change in mean-curvature (Bouaziz et al 2014, § 5.4).
Closeness	Constrains a point to a prescribed position.
Line, Plane, Circle, Sphere, Rectangle, Parallelogram	Constrains points to a lie on a geometric primitive (Bouaziz et al 2012, § 3).
Uniform Laplacian	Constrains a point to the average of its neighbors (Bouaziz et al 2012, § 4).
Uniform Laplacian of Deformation	Constrains a deformation vector with respect to the initial position to be the average of its neighboring deformation vectors (Bouaziz et al 2012, § 4).
Similarity	Constrains points to be similar (related by a rigid motion and uniform scaling) to one of the prescribed set of points. The similarity constraint automatically selects the closest of the prescribed sets of points to project to at each iteration (Bouaziz et al 2012, § 3.2).
Rigid	This constraint is equivalent to Similarity, only that it does not allow for uniform scaling (Bouaziz et al 2012, § 3.2).
Angle	Bounds the angle formed by three points (Deng et al 2015, § 3.3.2).

While many of the constraints intuitively apply to specific primitives, some of them can be applied to an arbitrary set of points defining novel shape spaces. For example, the circle constraint was often applied to all quads of a mesh because such circular meshes have desirable offset properties (Pottmann et al 2007). However, it can also be applied to each grid line of a quad mesh, defining an interesting shape space as illustrated in Fig. 5. Also note that the ShapeOp solver has no explicit knowledge of a mesh, but only of a list of points. This allows to apply ShapeOp to any set of points, e.g. mixing different geometric primitives such as splines, tetrahedral meshes, Bézier patches or triangle soups, that are parametrized by point positions.

5 Implementation

Our implementation of ShapeOp is distributed as a header-only, C++ library. While it is possible to develop C++ plugins for computational design environments this requires a larger and substantially more involved development investment than what is offered by higher level programming languages provided in .NET compliant CAD environments such as Rhino 3D and Revit. Here languages such as C#, VB and Python make development of computational design models fast, responsive, and interchangeable.

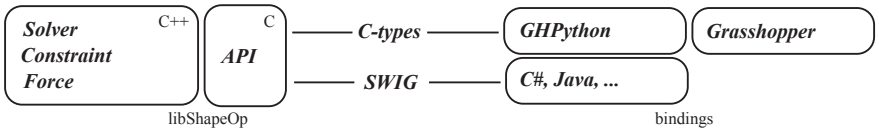

Fig. 2 Schematic overview of our implementation of ShapeOp.

Our implementation can be conceptually divided into two components: The core library `libShapeOp` with various bindings, and `applications` which use `libShapeOp`. The core library contains the abstract C++-classes `Solver`, `Constraint` and `Force`, and many classes deriving from and implementing them. The C-API provides an interface using C only, which simplifies calling `libShapeOp` from other code or programs considerably. ShapeOp also provides everything necessary to use SWIG (www.swig.org), a software development tool that can generate a multitude of bindings for `libShapeOp`. The applications contain the Grasshopper definitions using `libShapeOp`. The definitions use GhPython (www.food4rhino.com/project/ghpython) to enable Python in scripting components. Inside the component we use Python's ctypes to directly call `libShapeOp`.

There are six ShapeOp Grasshopper components in the current release. The functionality of each is implemented in a Python script. The `ShapeOp Constraint Solver` (`SOSolver`) is the central component and is the only one that calls the `libShapeOp` library. It sends points and constraint signatures to the library and retrieves the result. A constraint signature contains all the necessary information to setup a constraint: The constraint type represented by a string; the indices of points to be constrained with respect to the global list of points; the weight of this constraint; the scalars, a list of floating point numbers encoding additional settings of the constraint. `ShapeOp ConstraintSignature` (`SOCSig`) constructs the constraint signatures. Constraint signatures are implemented using a Python dictionary, so they could also be created by custom python components other than `SOCSig`.

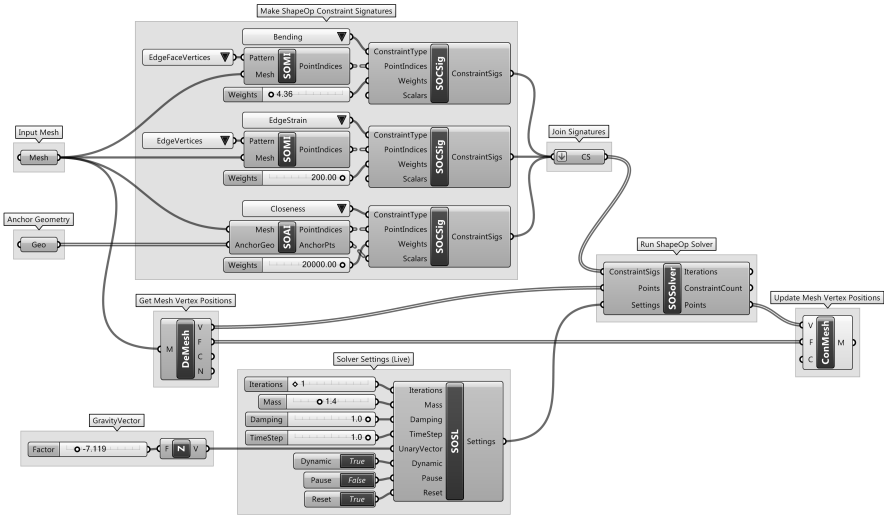

Fig. 3 The Grasshopper definition used for the hanging cloth example seen in Fig. 4.

6 Rhino/GhPython Implementation Examples

In Figs. 4 to 11, we provide some examples of using ShapeOp in Rhino/GhPython for different applications, including physics simulation, constrained modelling, rationalization, and form-finding.

7 Design Process

We believe that ShapeOp can have a considerable impact on a design process in various ways. In interactive modelling tools the graphical user interface often cost non-negligible fraction of the execution time, in particular on large data. Since ShapeOp is built modularly, it can be run independently of any user interface. Also, due to the C/C++ implementation, ShapeOp runs natively and is heavily optimized by compilers and parallelization with OpenMP (www.openmp.org). ShapeOp can therefore potentially handle huge models.

In ShapeOp the global and local steps are both numerically stable least-squares problems, implying that the overall method is also stable and robust. Also, many constraints such as the plane constraint only concern the relative arrangement of points and stay satisfied after applying a translation to all points. ShapeOp utilizes this in the global step by implicitly solving for the translations for each constraint independently. This allows for constrained points to move arbitrarily far and greatly increases convergence speed.

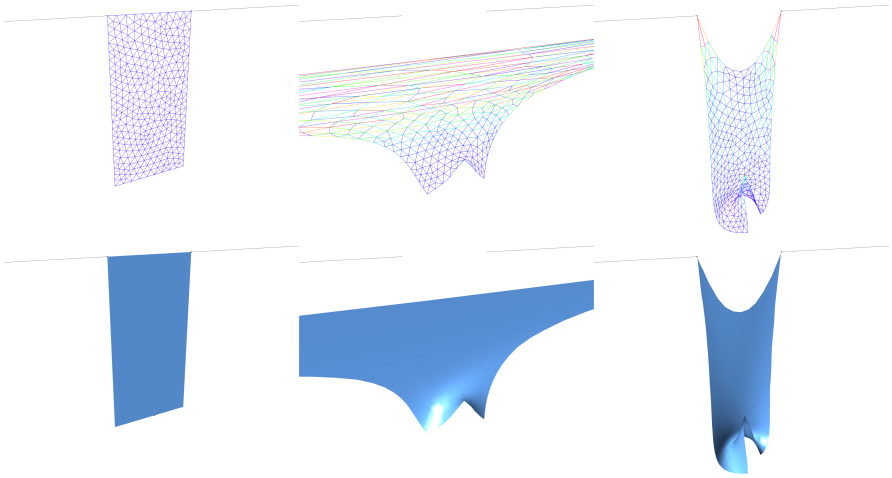

Fig. 4 Use of ShapeOp for physics simulation of elastic materials. A hanging cloth is modelled using edge strain and bending constraints. The three vertices are anchored using closeness constraints and all points are subjected to a gravity load. Left: The input mesh. Middle: The constrained mesh at the first solve iteration in which the anchors are immediately moved very far apart. Right: The constrained mesh after 100 iterations with the anchor point moved back to their starting positions. Top: Wireframe rendering with the edges coloured by their strain (red = high, blue = low). Bottom: Shaded rendering. The example demonstrates both the stability and the fast convergence of the solver.

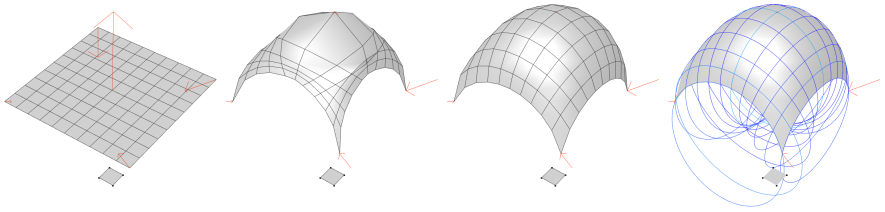

Fig. 5 Use of ShapeOp for constrained modelling of a shell with rational geometric properties. The vertices on the parameter lines of a quad-mesh are constrained to always lie on a circular arc using the circle constraint. Each face is constrained towards being square using the similarity constraint. Five vertices are anchored to different positions than their initial positions, enabling shape exploration. Left to right: 1) The input mesh, the face used for similarity and vectors visualizing start/end positions for the anchors. 2) The constrained mesh after 10 iterations. 3) The constrained mesh after 300 iterations. 4) The constrained mesh with circles drawn through each of the parameter line vertices (Red = Line vertices distance to circle is large, Blue = Line vertices distance to circle is small).

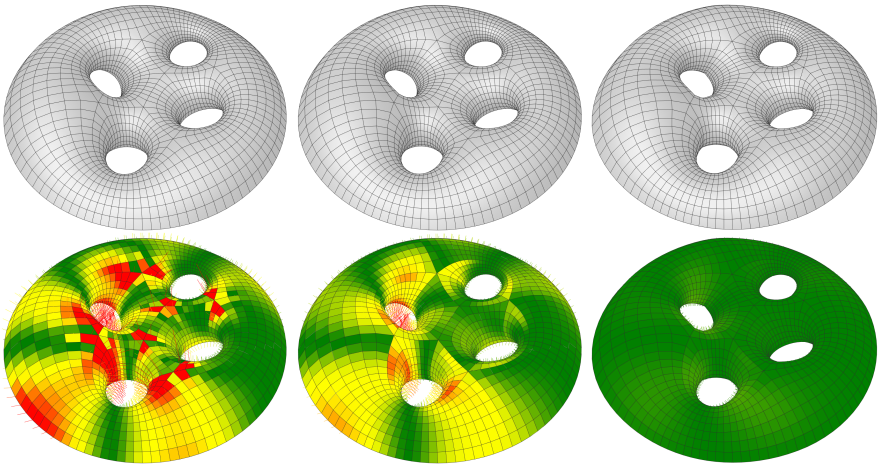

Fig. 6 Use of ShapeOp for rationalizing an existing geometry. Each face of the quad-mesh is constrained towards being planar using the plane constraint. Each vertex is constrained to its initial position using the closeness constraint by a small weight to maintain the shape of the mesh. Left: Input mesh. Middle: The constrained mesh after 10 iterations. Right: The constrained mesh after 200 iterations. Top: Shaded rendering. Bottom: Planarity analysis rendering (Red = Low planarity, Green = High planarity).

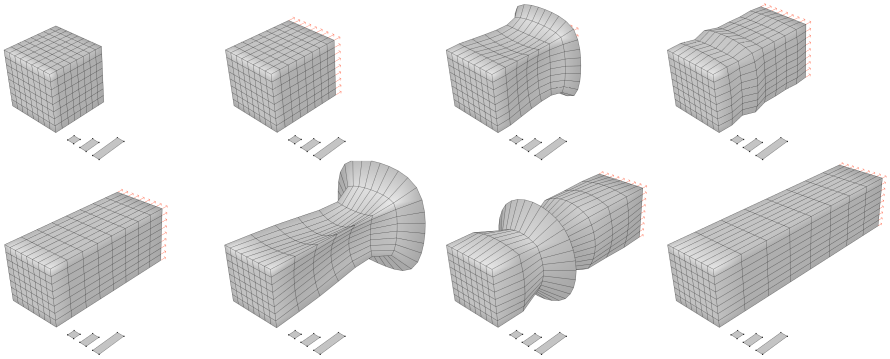

Fig. 7 Use of ShapeOp for constrained modelling of box shape with multiple rigid shape targets. A quad-mesh box is anchored at the vertices on two sides of the box. The image sequence shows the vertices on one side being pulled away over time. As this occurs each mesh face attempts to project itself onto one of the three shape targets below the box. The solver has been initialized using dynamics leading to the rippling effect as the faces switch their projection targets from short to medium to long. This projection type is enabled using the rigid constraint.

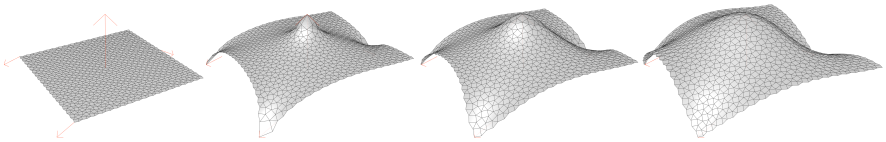

Fig. 8 Use of ShapeOp for constrained modelling of a shell with topologically different shape targets. A planar mesh composed of both triangles and quads is anchored at four vertices using the closeness constraint. Using the similarity constraint, each face is constrained towards their initial shape i.e. an equilateral triangle or a square. 1) The input mesh. 2) The mesh after 1 iteration. 3) After 100 iterations. 4) After 500 iterations.

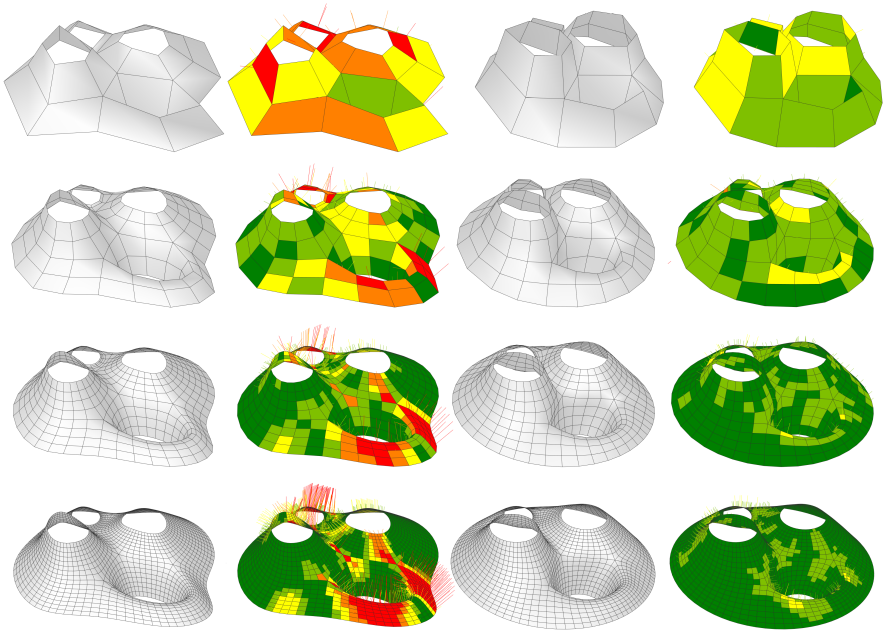

Fig. 9 Use of ShapeOp for constrained modelling of a randomly generated quad-mesh with multiple constraints as design drivers. The example demonstrates the effect of applying the same constraints on meshes with different resolutions. It uses three primary constraints: 1) Limit the internal angles of each face to be within 80 and 110 degrees, 2) The boundaries of the mesh should lie on circles, 3) Each face should preserve its area. Additionally, a Laplacian of displacement constraint is added which smoothens out the mesh while maintaining the shape, and a bending constraint is added which ensures that face-face angles do not become too acute. The color code is a based on scoring system: The internal angles for each face are calculated. If an angle is within the desirable range it is scored 0, else 1. The scores are added for each face, best face score is 0 (Dark green) worst is 4 (Dark red).

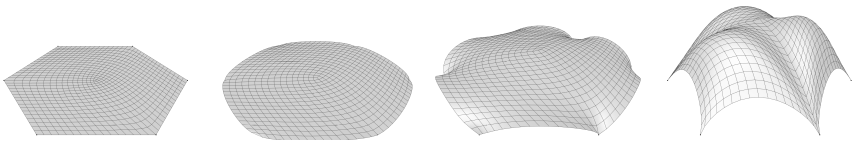

Fig. 10 Use of ShapeOp for funicular form finding. A hexagonal quad-mesh is anchored at each corner and subjected to an inverse gravity load. In this image sequence the only other constraint is that each edge should be 2.0 units long. This is implemented using the edge strain constraint. 1) The input mesh. 2) The constrained mesh after 1 iteration. 3) The constrained mesh after 10 iterations. 4) The constrained mesh after reaching equilibrium at iteration 1000.

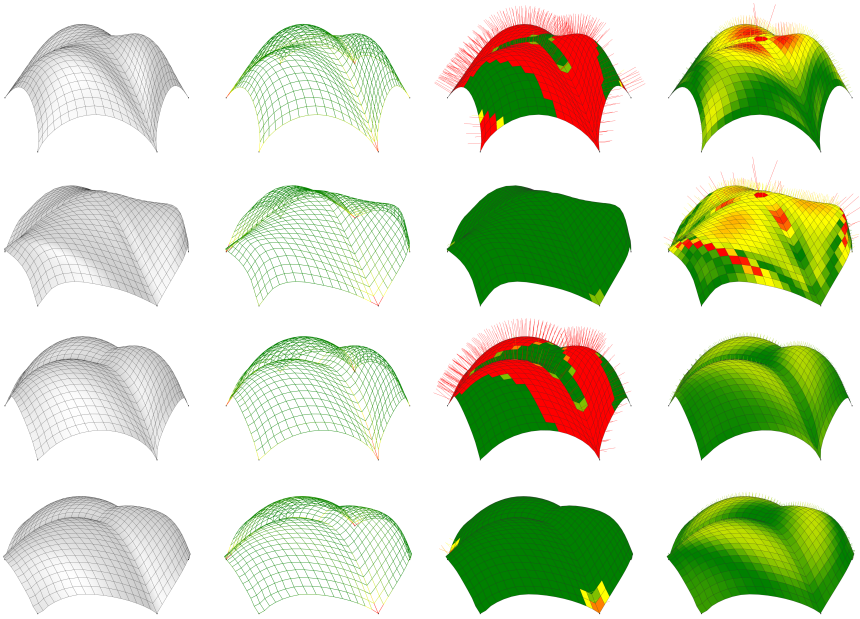

Fig. 11 Use of ShapeOp for funicular form finding under fabrication constraints. Demonstrates the effect of combining different constraints: Desired edge length, planarity of faces and desired range of internal face angles. All images show the constrained mesh at equilibrium. From left to right: 1) Shaded rendering. 2) Edge length deviation from desired length. 3) Face angles deviation from desired angle range. 4) Face Planarity Deviation (Red = High deviation, Green = Low deviation). Row 1: The mesh with edge strain constraints. Row 2: The mesh with edge strain and internal mesh face angles constraints. Row 3: The mesh with edge strain and face planarity constraints. Row 4: The mesh with edge strain, internal mesh face angles and face planarity constraints.

8 Future Work

As future work we plan to further explore the combination of continuous and discrete constraints. It is a common feature of many design problems to have some design components that can only be selected from a finite set of choices. Unfortunately, such a finite set can be too restrictive to preserve the aesthetic quality of the design. In this case, enhancing the optimization to automatically detect a sparse set of constraints that needs to be violated in order to better preserve the design intent would lead to a much richer design exploration solution.

9 Conclusion

We believe that ShapeOp is a promising candidate for state-of-the-art computational geometric design and evaluate its potential in this paper. We explain the theoretical advantages over existing methods, and present the implementation as a simple, fast and extensible C++ library (www.shapeop.org). Our examples use the scripted grasshopper components provided with ShapeOp to highlight its practical importance.

Acknowledgements We thank the reviewers for their valuable comments. This work has been supported by Swiss National Science Foundation (SNSF) grant 200021_137626 and the Danish Council for Independent Research (DFF). This research has received funding from the European Research Council under the European Unions Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement 257453, ERC Starting Grant COSYM.

References

Attar R, Aish R, Stam J, Brinsmead D, Tessier A, Glueck M, Khan A (2009) Physics-based generative design. In: CAAD Futures Conference, pp 231–244

Bouaziz S, Deuss M, Schwartzburg Y, Weise T, Pauly M (2012) Shape-up: Shaping discrete geometry with projections. Computer Graphics Forum 31(5):1657–1667

Bouaziz S, Martin S, Liu T, Kavan L, Pauly M (2014) Projective dynamics: Fusing constraint projections for fast simulation. ACM Trans Graph 33(4):154:1–154:11

Day AS (1965) An introduction to dynamic relaxation. The Engineer 219:218221

Deng B, Bouaziz S, Deuss M, Kaspar A, Schwartzburg Y, Pauly M (2015) Interactive design exploration for constrained meshes. Computer-Aided Design 61(0):13–23

Kilian A, Ochsendorf J (2005) Particle-spring systems for structural form finding. Journal of the International Association for Shell and Spatial Structures 46(2):77–84

Nocedal J, Wright S (2006) Numerical Optimization, 2nd edn. Springer Series in Operations Research and Financial Engineering, Springer-Verlag New York

Piker D (2013) Kangaroo: Form finding with computational physics. Architectural Design 83(2):136–137

Pottmann H, Liu Y, Wallner J, Bobenko A, Wang W (2007) Geometry of multi-layer freeform structures for architecture. ACM Trans Graph 26(3)

Pottmann H, Eigensatz M, Vaxman A, Wallner J (2015) Architectural geometry. Computers & Graphics 47(0):145 – 164

Senatore G, Piker D (2015) Interactive real-time physics: An intuitive approach to form-finding and structural analysis for design and education. Computer-Aided Design 61(0):32–41

Tang C, Sun X, Gomes A, Wallner J, Pottmann H (2014) Form-finding with polyhedral meshes made simple. ACM Trans Graph 33(4):70:1–70:9

Witkin A, Baraff D (1997) Physically based modeling: Principles and practice. Siggraph '97 Course notes